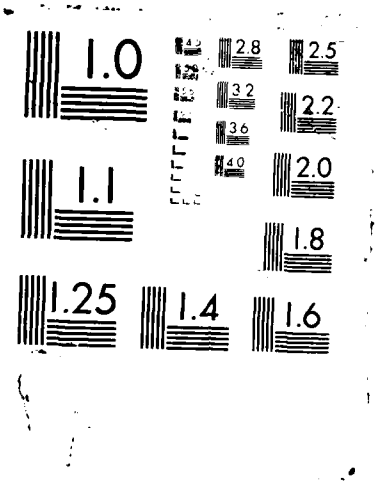
AD-A190 557 AN INQUIRY INTO THE BENEFITS OF MULTIGAUGE PARALLEL COMPUTATION(U) WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE L SNYDER MAY 85 TR-85-06-03 N00014-85-K-0328 1/1

UNCLASSIFIED F/G 12/6 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: none	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): An Inquiry into the Benefits of Multigauge Parallel Computation	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Lawrence Snyder	8. CONTRACT OR GRANT NUMBER(s): N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Washington, Department of Computer Science, Seattle, Washington 98195	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Arlington, VA 22217	12. REPORT DATE: May 1985
	13. NUMBER OF PAGES: 5
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel programming; multigauge parallel computer; gauge shifting; multitier algorithms; MIMD, SIMD

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A multigauge parallel computer is a machine whose processor elements can be partitioned into distinct processors with narrower data paths. ILLIAC IV was a multigauge machine. This paper addresses the question of whether multigauge computation is useful. First, the concept is defined. Next it is argued that multigauge machines offer truly new computational facilities, rather than being other architectures in disguise. Finally, a class of algorithms is identified that can exploit multigauge architectures and an example is presented to illustrate multigauge machine performance improvements.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

An Inquiry into the Benefits of Multigauge Parallel Computation

by
Lawrence Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195
TR-85-06-03

This paper has been funded in part by the Office of Naval Research Contract No. N00014-85-K-0328 and National Science Foundation Grant No. DCR-8416878.

AN INQUIRY INTO THE BENEFITS OF MULTIGAUGE PARALLEL COMPUTATION

Lawrence Snyder

Department of Computer Science
University of Washington
Seattle, Washington 98195

Abstract–A multigauge parallel computer is a machine whose processor elements can be partitioned into distinct processors with narrower data paths. ILLIAC IV was a multigauge machine. This paper addresses the question of whether multigauge computation is useful. First, the concept is defined. Next it is argued that multigauge machines offer truly new computational facilities, rather than being other architectures in disguise. Finally, a class of algorithms is identified that can exploit multigauge architectures and an example is presented to illustrate multigauge machine performance improvements.

Introduction

In parallel computation speed comes from organizing many processors to solve a single problem, so it is natural to think of accelerating a parallel computer by adding more processors rather than by speeding up those currently in use. But speeding up the processor elements (PEs) is an effective way to improve performance. For example, a factor of two improvement in PE speed yields a factor of two improvement in instruction executions per second and this can often be done with only a modest amount of extra hardware; achieving the same improvement by adding PEs requires at least twice the hardware. (*Utilizing* the performance improvement has its problems with either solution: Faster PEs cause memory latency to have a greater effect on observed performance, and more PEs exacerbate communication bottlenecks.) Clearly, making faster PEs is only a tactic in the battle for improved parallel computer performance because there is a limit to how fast a sequential processor can get, and the greater the speed of a PE, the greater the cost of improving on it. Providing more PEs is the strategy that will win in the long run. For any given situation, however, the question is: more PEs or faster PEs?

One technique with elements of both approaches is to introduce multiprocessing into the PEs. The technique, called *gauge shifting*, exploits the fact that data types come in different sizes and the smaller ones might be processed concurrently by partitioning the data path. The first machine capable of gauge shifting was ILLIAC IV [1]; the 64 64-bit PEs could also be used as 128 32-bit PEs or as 512 8-bit PEs. [1] Although some programs were written for ILLIAC IV using the 32-bit gauge PEs, the machine was apparently never used in the way proposed here, namely to shift back and forth between different gauges dynamically.

[1] Whether one thinks of an n processor machine with 64-bit PEs as becoming a $2n$ processor machine with 32-bit PEs, or becoming an n processor machine with dual 32-bit PEs, depends on other aspects of the architecture as described in the second section.

Funded in part by the Office of Naval Research Contract No. N00014-85-K-0328.

The purpose of this paper is to inquire into the benefits of the general idea of gauge shifting as a means of improving parallel computer performance. The presentation includes a more precise definition of the concept, the identification of a class of algorithms capable of exploiting dynamic gauge shifting, and an investigation as to whether gauge shifting constitutes a fundamentally different form of computation.

Definitions and Background

In this section the concepts alluded to in the introduction are made more precise and relevant related work is cited.

A *multigauge architecture* is a sequential computer with a data path width of B bits, called the *wide track* machine, which can be partitioned into k distinct sequential machines, called *narrow track* machines, each with a $\lfloor B/k \rfloor$-bit wide data path. It is convenient to permit a von Neumann machine to be a trivial (i.e. $k=1$) multigauge machine, and the term *dual track* will be used to refer to the case where only one nontrivial value of k is implemented.

Notationally, B will denote the wide track width, b will denote the narrow track width, and it is assumed hereafter that $b \cdot k = B$. The multigauge machine can be described by listing the different track widths it supports. Thus the ILLIAC IV PEs would be (64, 32, 8) multigauge machines.

The instructions executed by the narrow track machines can form either a single stream, i.e. an *SIMD multigauge architecture*, or multiple streams, i.e. an *MIMD multigauge architecture*, but there are some pragmatic limits. For example, it seems unrealistic when $b = 1$ to postulate MIMD execution since fetching separate instructions for each bit of the data path, decoding them, calculating operand addresses, and fetching disparate bits from memory is excessive effort for the amount of computation being performed. So, postulate the *MIMD threshold*, the number of bits wide a narrow gauge machine must be before MIMD execution is "justified". Here the MIMD threshold will be taken to be eight bits; shifting to gauges narrower than eight bits will be assumed to be SIMD execution.

A *multigauge parallel computer* is a parallel machine whose PEs are capable of gauge shifting. There are two ways to implement this capability: The machine with wide track PEs and the machine with narrow track PEs are each instances of the same architectural family, i.e. if the size of the PEs is ignored, the narrow track machines appear to be versions of the wide track architecture scaled up to more PEs. Alternatively the architectural relationships do not change as a result of shifting except that the PEs become small multiprocessors. The former are referred to as Type A multigauge parallel machines and the latter as Type B multigauge parallel machine. To illustrate, an 8×8 mesh-connected architecture of 64-bit PEs that shifts to a 16×16 mesh-connected architecture with 16-bit PEs is a Type A multigauge parallel computer. Alternatively, if the machine

remains an 8 × 8 mesh but each PE becomes a quad processor, then this would be a Type B multigauge parallel architecture.

The multigauge concepts discussed here are reminiscent of several previous studies. The "Dynamic Architecture" of Kartachev and Kartachev [2] is based on connecting together basic narrow track (e.g. 16-bit) computers on a bus to achieve a wider word width. The wide track machine, called a *dynamic computer group*, can be simultaneously shifted into different gauges as long as the narrow track machines each have a multiple of 16-bit data path width. The concept of switching between SIMD and MIMD execution modes has been most fully developed in the Partitioned Array SIMD-MIMD (PASM) Computer of H. J. Siegel and his colleagues [3]. PASM uses the same fixed size processor elements in both modes. The Very Long Instruction Word Architectures of Fisher [4] utilize several independent, fixed gauge ALUs that are neither split or joined. The Content Addressable Array Processor (CAAP) of Weems *et al.* [5] couples several narrow gauge machines with a wide gauge machine; the machines are distinct rather than being restructurings of the same hardware. Similarities with other architectures undoubtedly remain to be explored.

To close this section notice that certain other implementations besides ILLIAC IV provide some degree of gauge shifting. For example, the Cyber 205 [6] can partition the 64-bit data path into two 32-bit data paths.

An Analysis of Benefits

Multigauge computers appear to offer a benefit over fixed gauge machines on computations involving small size data types, since k narrow gauge machines provide k-fold parallelism. But because the multigauge idea uses essentially the same hardware with only modest enhancements, one wonders if perhaps the speedup is only an illusion. This concern is further strengthened by the observation that k 1-bit machines each performing an *AND* is essentially equivalent to a k-bit fixed gauge machine executing a bit-wise *AND*. Therefore, it is necessary to argue that multigauge computation is a fundamentally different phenomenon, and we will. In addition, we will identify the exact source of the improved performance.

Before beginning, we make some preliminary observations. First, it is appropriate to limit our arguments to multigauge machines as opposed to parallel multigauge machines, since we are interested in the multigauge phenomenon alone, and the arguments either extend directly to the parallel case or become more complicated due to interactions with other parts of the parallel architecture. Second, we will assume that multigauge machines have comparable performance to like gauge sequential computers. This is a significant assumption because there is somewhat greater complexity with a multigauge machine, and so we are assuming that it is completely transparent during wide track execution. (We also assume, though perhaps somewhat less realistically, that the narrow gauge instructions run at the same rate.) Another reason why comparable performance is a significant assumption is that there are many strategies for speeding up sequential machines, and these may not be compatible with the multigauge approach (or each other for that matter). So adopting a multigauge design may preclude other optimizations. Still, comparable performance is a plausible assumption to get us started; if fundamental benefits can be identified, the detailed design needed to resolve these other issues will be justified.

To understand how performance gains might accrue from gauge shifting, compare the narrow track machines executing in SIMD mode with a standard sequential machine. For the comparison to be interesting suppose that instructions exist for the sequential machine so all data types of small size can be treated like the bit-wise *AND*s mentioned above. Specifically, a standard k·b-bit sequential machine having $(k-1)b$ unused bits when computing on b-bit data has its instruction set extended to support k b-bit operations elementwise within a word. For example, in addition to logicals one might have instructions to do two half word *ADD*s, etc. Such an extended sequential machine would not be equivalent to a multigauge machine executing even in SIMD mode because, although there is one instruction stream in force in both machines and multiple data values being manipulated in both, there is but one address for each operand group of the extended sequential machine. Data values must be packed together in a word to achieve k-way parallelism. There is no such restriction for the multigauge machine. [2]

For the two machines to have equal performance requires that every algorithm using differently addressed data streams of narrow width data be convertible into a packed form that can be referenced by a single address stream. This seems to be extremely unlikely. The point of the comparison is twofold: The bitwise *AND* is really a special case rather than being a good example of the multigauge idea, and although construction of a multigauge machine will engender certain costs associated with supporting multiple operand fetching, the feature has apparent benefit even in the SIMD case.

Having focussed on multiple operand fetching, we now address the benefits of multiple instruction fetching. (The argument amounts to a defense that MIMD computation is more powerful than SIMD computation.) Postulate a sequential machine capable of packing several operator/operand specifications together in a single instruction word. Such a machine, though still with only a single program counter, would be able to execute several distinct programs provided that a particular condition could be enforced on the execution sequence, namely, that they remain "in unison". In particular, let

$I_1, I_2, \cdots, I_n$ /*program I*/
$J_1, J_2, \cdots, J_n$ /*program J*/
...
$K_1, K_2, \cdots, K_n$ /*program K*/

be programs. As long as these are straight line code, the instructions and their operands can be packed together in instruction words,

$< I_1, J_1, \cdots, K_1 >$
$< I_2, J_2, \cdots, K_2 >$
...
$< I_n, J_n, \cdots, K_n >$

and be executed by a machine with a single program counter. If there is a conditional branch, say in J_1 with target instructions J_2 and J_2', then we need to provide another sequence of packed instructions

[2] It may be that once the data types get too narrow, addressing restrictions must apply for the same reasons motivating the MIMD threshold. If the limitation is to a single data stream, the machines could be equivalent.

$$< I_2, J_2', \cdots, K_2 >$$
$$\cdots$$
$$< I_n, J_n', \cdots, K_n >$$

so that the branching can be provided for and still remain "in unison".

Obviously, storing the whole execution sequence is unrealistic, so we concentrate on storing short segments that represent the instructions that could be executing concurrently. Consider programs composed of short blocks which each test a value, change it and then jump to one of two different blocks depending on the outcome of the test. Each program will jump around to different locations in an unpredictable order. When we consider the programs together we see that any given instruction could be executing with any combination of instructions from the other programs. Thus to generate a program that can be executed with a single instruction counter requires that essentially all tuples of instructions, one from each program, must be provided for.

Returning now to the question of multigauge computation being faster than an equivalent sequential computation, we note that programs of the type just discussed can be stored in $O(kn)$ space on a multigauge machine but will require $O(n^k)$ space on a sequential machine. This disparity is too great to be the basis of a fair comparison, as can be seen by instantiating the functions for realistic size values such as $k=4$ and $n=100$. Assuming the sequential machine is limited to a comparable amount of space, it must cease to exploit packed instructions and thus be reduced to executing the programs (essentially) separately. The resulting longer execution times imply the existence of a fundamental performance improvement with multigauge computation.

The conclusions from the preceding discussion are that multigauge computation is fundamentally different from sequential computation and that potential performance improvements exist. Fetching multiple operands and multiple instructions, though complicating to the machine design, have been shown to be a source of power. Whether the benefits can actually be realized in a physical design is an interesting and challenging open problem.

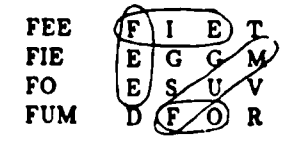
Since it would seem that a multigauge architecture will essentially be many program counters, control units, instruction decoders, etc., sharing a data path and a memory, neither of which is a very scarce resource, it is evidently not the case that gauge shifting is justified on purely economic grounds. It is, therefore, appropriate to close this section with a brief philosophical discussion of additional benefits of multigauge (parallel) computation. One advantage is that multigauge machines neutralize a rather pointless argument about the merits of "coarse grain" versus "fine grain" computation; these machines can be either, as appropriate. More importantly, multigauge architectures respond to the fact that certain problems display several types of computational needs - voluminous but rather direct data manipulation followed by much more complex, sophisticated processing. (A more detailed description is given in the next section.) The key point is that multigauge machines can do both with respect to the same memory. It is not that memory is expensive, but rather that data occupancy is. Once in memory, data should be processed where it resides rather than being moved about, unchanged and thus introducing overhead. This aspect is extremely important for nonshared memory architectures. Finally, there are the esthetics of being able to describe directly different gauge computations rather than encoding one in another. Of course, no matter how elegant, a machine doesn't count for much unless it is useful for some important problems; so we consider algorithms that can exploit gauge shifting.

Two Tier Algorithms

Although we have concentrated on the architectural issues of gauge shifting, the motivation for studying the phenomenon, as indicated in the last section, is to support the execution of certain kinds of algorithms, the general class of which we call *multi-tier algorithms*.[3] The simplest members of the algorithmic class are *two tier* algorithms which have the property that there is an enormous amount of simple data processing on small size data items, followed by more complicated processing on more complex data structures. Problems requiring two tier algorithms for their solution arise in many applications areas such as artificial intelligence, data bases and image processing. For example, in image processing the first tier would involve pixel level processing where regions of two images might be correlated to register the two pictures. The higher tier processing focusses on such activities as motion detection.

Two tier algorithms are ideal for execution on a multigauge computer. The narrow gauge processing of the first tier can benefit from the greater parallelism, while the wide track mode supports the more complex processing of the higher tier. It would be unrealistic to present a two tier algorithm for a true application since the higher tier would be complex beyond what is necessary for illustration. However, we can present an algorithm to solve a simple puzzle, Word Find, which illustrates the principal concepts of switching between different gauges.

Word Find is a common puzzle in which the solver is presented with an $m \times m$ array of letters, A, and a word list W of size $r \times s$, i.e. there are r words, the longest of which is s letters. The object of the puzzle is to locate the words of the list in the array of letters as consecutive positions in a row, column or diagonal. For example,



Finding the words will be the first tier problem. The words will be tested to see if they are all found exactly once, making the higher tier processing to find if all the words of the list exist in the array without duplicates.

The Word Find problem will be solved on a Type A (32, 8) multigauge parallel machine of the CHiP architecture [8]. Recall that Type A machines display the same architecture in both wide and narrow tracks, so in the present case both gauges are assumed to be configurable. We will use an eight way mesh interconnection for the narrow gauge and a binary tree interconnection for the wide gauge. Thus, each tree node will correspond to a 2×2 mesh subarray of the narrow gauge.

To simplify the presentation, we make some assumptions. First we assume that the letter array A is already loaded into the processor array, one latter per narrow gauge PE. Second, each narrow gauge PE has access to the $r \times s$ word list W.

[3] These algorithms have also been called *hierarchical* [7]

which means there is at least one copy per wide track machine; there may be one per narrow track depending on how memory reference conflicts are handled. Third, we simplify matters by only searching for horizontal matches; the other cases are trivial extensions. Finally, we ignore "edge effects", i.e. we do not worry about the case where the right column PEs have no right neighbor.

The $r \times s$ word list W has a special form. The words are *right* justified, padded (on the left) with blanks (b), augmented by a blank column ($= 0$) on the left and a blank row ($= r+1$) on the bottom. The words are lexicographically sorted by right-most position, i.e. words *ending* in a "a" come first. (See Figure 1.) Also if W_{ij} is a nonblank character and $W_{i-1,j} = W_{ij}$ for this and all larger values of j then W_{ij} is replaced by a ditto mark ("). Finally, a bit vector *find* [1:r], initially zero, is local to each narrow gauge PE and is assigned 1 in the i^{th} position if this PE is the first character of an (horizontal) instance of the i^{th} word.

The matching part of the algorithm uses s iterations to locate which of the r s-length words match. During the j^{th} iteration a PE reads a value (p) from the east indicating either that no word matches in the last $j-1$ positions ($p = 0$), or giving the index of the (first) word in the list that matches in the last $j-1$ positions ($p \neq 0$). If the match had failed or fails this time, the $p = 0$ value is sent west. If the match continues a $p \neq 0$ is sent west. If it happens that a match also succeeds, this is recorded in the *find* vector. Finally, the match that had been found could fail, but because of the ditto marks (indicating other words with the same suffix) the index could be moved to a subsequent word.

We give the text of the narrow gauge program as if for the Poker parallel programming environment [9].

```
code match;
/* The information global to this process is:
 character A          The element of
                      the word find
                      letter array stored
                      in this narrow
                      track PE.
 character array      W[1..r+1,0..s]
                      The word list,
                      padded and right
                      justified
 integer r, s         The word list
                      size, i.e. num-
                      ber of words and
                      maximum num-
                      ber of letters.
*/
ports East, West;
begin integer i,j,p; Boolean array find [1..r];
/* locate word matching A in last character */
p := 0; /* initialize to "none found" */
for i := 1 to r do
    if W[i,s] = A then { p := i;
        if W[i,s-1] =' b' then find[i] := 1;
        go to L1 };
L1 : West ← p; /* send index to neighbor */

/* general matching - next to last through first character */
for j := s - 1 to 1 step - 1 do
begin p ← East; /* receive index from neighbor */
if p ≠ 0 then
    begin
        if W[p,j] = A ∧ W[p,j-1] =' b' then find [i] := 1;
    L2 : if W[p,j] ≠ A ∧ W [p+1,j+1] = ' " '
            then { p := p+1; go to L2};
        if W[p,j] ≠ A then p := 0
    end;

West ← p /* send index to neighbor */

end
end;
```

At the completion of the narrow gauge programs, the machine changes state and begins to execute the wide track program.

The wide track program uses a binary tree interconnection of PEs. Each node refers to the *find* vectors of its four constituent PEs, treating the values as words and using logical bitwise operations on them. (The careful reader will recognize that our use of bitwise *AND*s here is only a coincidence and has nothing to do with the discussion in the third section.) The goal is to recognize if all and only the words of the word list appear in the letter array, so each node " merges" its *find* vectors; if it is a leaf it passes the result to its parent, and if it is a nonleaf it "merges" in the results from its two children before passing the result to its parent. To perform the "merge" operation, we use a function *merge* that checks for and records any collisions and the unions the bit sequences together. At the end, the outcome of the collision tests is passed up the tree.

The code for the wide track program of nonleaf node is given below. Leaf programs would not have the starred lines.

```
code combine;
/* The information global to this process is:
 Boolean array  PE1.find,  Find arrays
                PE2.find,  from the narrow
                PE3.find,  gauge PEs
                PE4.find
 integer r                 Number of bits in finds
*/
ports leftchild, rightchild, parent;
begin
   integer i, ans, r, l;
   logical temp, temp1, temp2;
   logical array   PE1bits [1..⌈r/32⌉],
                   PE2bits [1..⌈r/32⌉],
                   PE3bits [1..⌈r/32⌉],
                   PE4bits [1..⌈r/32⌉];

/* make data value correspondence */
equivalence (PE1.find, PE1bits), (PE2.find, PE2bits),
       (PE3.find, PE3bits), (PE4.find, PE4bits);
function C(a,b); logical a,b;
{ if(a ∧ b) ≠ 0 then ans ← 0; C := a ∨ b };
ans:=1;
```

```
for i := 1 to ⌈r/32⌉ do
    begin
        temp1:= C(PE1bits, PE2bits);
        temp2:= C(PE3bits, PE4bits);
        temp:= C(temp1, temp2);
        temp1 ← leftchild; temp 2 ← rightchild;    /* * */
        temp:= C(temp, temp1);        /* * */
        temp:= C(temp, temp2);        /* * */
        parent ← temp
    end;
l ← leftchild;        /* * */
r ← rightchild;       /* * */
ans:= ans × l × r;    /* * */
parent ← ans
end
```

The "parent" of the root, presumably the controller, receives the results.

For the algorithm analysis, we notice that as long as the W array bounds remain within the "single precision" range of the narrow track PEs, there is essentially full speedup for the *narrow track* phase of computation. The parallelism in the wide track phase is restricted to that provided by multiple PEs rather than gauge shifting. Thus, we have for some $C_1 > 0$ and $C_2 > 0$,

$$C_1(r+s)m^2 + C_2 r \log m$$

steps. If we suppose that $B = 32$ and $k = 4$ then we achieve essentially the whole factor of four speedup on the work represented in the first term. Since this is the dominate term in the computation, the benefit applies to the whole algorithm. Moreover, if the problem size grows in terms of m the benefits persist.

Conclusions

The goal of this paper has been to inquire into the benefits of gauge shifting. Towards this goal we have defined multigauge architectures and related concepts. We have argued that multigauge computation represents a fundamentally different kind of computation, not simply sequential computation in disguise. Finally, we have identified a class of algorithms, two tier algorithms, that can exploit gauge shifting.

The benefits, analyzed in the abstract, seem to be substantial, suggesting the worth of a design and implementation effort to identify and quantify the problems.

Acknowledgements

It is a pleasure to thank Jean-Loup Baer and Janice E. Cuny who read copies of an early draft of this paper and provided very helpful comments. I would also like to thank Kye S. Hedlund with whom early precursors to these ideas were discussed.

References

[1] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh and D. L. Slotnick, "The Illiac IV System," Proceedings IEEE, 60(4), 1972, pp. 369-379.

[2] S. I. Kartashev and S. P. Karashev, "A Multicomputer System with Dynamic Architecture," IEEE Transactions in Computers, C-28(10), 1979, pp. 704-721.

[3] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr. and S. D. Smith, "PASM: A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition," IEEE Transactions on Computers, C-30(12), 1981, pp. 934-947.

[4] Joseph A. Fisher, Very Long Instruction Word Architectures and the ELI-512, Department of Computer Science, Yale University, YALEU/DCS/RR-253, (April, 1983)

[5] C. Weems, S. Levitan, D. Lawton and C. Foster, A Content Addressable Array Parallel Processor and Some Applications in Image Understanding, Scientific Applications, Inc., SAI-84-176-WA, (1984).

[6] Control Data Corporation, CDC Cyber 200 Model 205 Computer System: Hardware Reference Manual, Control Data Corporation, St. Paul, (1981).

[7] Janice E. Cuny, personal communication.

[8] L. Snyder, "Introduction to the Configurable Highly Parallel Computer," Computer, 15(1) January, 1982, pp. 47-56.

[9] L. Snyder, "Parallel Programming and the Poker Programming Environment," Computer, 17(17) July, 1984, pp. 27-36.

END
DATE
FILMED
4-88
DTIC